软件学报 ISSN 1000-9825, CODEN RUXUEW
[doi: 10.13328/j.cnki.jos.007347] [CSTR: 32375.14.jos.007347]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

GhostFunc:一种针对 Rust 操作系统内核的验证方法 *

何韬, 董威, 文艳军

(国防科技大学计算机学院,湖南 长沙 410073)

通讯作者: 董威, E-mail: wdong@nudt.edu.cn

摘 要: 操作系统是软件的基础平台,操作系统内核的安全性往往影响重大.Rust 是逐渐兴起的内存安全语言,具有生命周期、所有权、借用检查、RAII 等安全机制,使用 Rust 语言构建内核逐渐成为当前热门的研究方向.但目前使用 Rust 构建的系统多包含部分 unsafe 代码段,无法从根本上保证语言层面的安全性,因而针对 unsafe 代码段的验证对于保证 Rust 构建的内核正确可靠尤为重要.本文以某使用 Rust 构建的微内核为对象,提出了 GhostFunc 的 safe 和 unsafe 代码段组合验证方法,将两类代码段采用不同层级的抽象,使用 GhostFunc 进行组合验证.本文针对任务管理与调度模块,基于 λ_{Rust} 形式化了 Arc<T>等 unsafe 代码段,并给出了形式化 GhostFunc 的具体实现,完成了此方法的验证实例.本文所有验证工作基于定理证明的方法,在 Coq 中采用 Iris 分离逻辑框架完成了正确性的验证.

关键词: 形式化验证;操作系统内核;分离逻辑;Rust;定理证明

中图法分类号: TP316

中文引用格式: 何韬, 董威, 文艳军. GhostFunc:一种针对 Rust 操作系统内核的验证方法. 软件学报. http://www.jos.org.cn/1000-9825/7347.htm

英文引用格式: He T, Dong W, Wen YJ. GhostFunc: A Verification Method for Rust Operating System Kernels. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/7347.htm

GhostFunc: A Verification Method for Rust Operating System Kernels

HE Tao, DONG Wei, WEN YJ

(National University of Defense Technology, Changsha 210094, China)

Abstract: The operating system serves as the foundational platform for software, and the security of its kernel is often of significant importance. Rust, a memory-safe language that has been gradually gaining popularity, incorporates safety mechanisms such as lifetimes, ownership, borrowing checks, and RAII. Building kernels using Rust has become a prominent area of research. However, systems constructed with Rust often contain some unsafe code segments, preventing the language from offering comprehensive guarantees of safety at the language level. Therefore, verifying these unsafe code segments is crucial for ensuring the correctness and reliability of Rust-based kernels. This paper proposes a method for combining the safe and unsafe code segments, called GhostFunc, to verify a microkernel built with Rust. The method applies different levels of abstraction to the two types of code segments and uses GhostFunc for the combination verification. Focusing on the task management and scheduling module, this paper formalizes unsafe code segments such as Arc<T> using λ_{Rust} and presents the formal implementation of GhostFunc. A verification example of this method is also provided. All verification work is based on theorem proving, and correctness is validated in Coq using the Iris separation logic framework.

Key words: Formal Verification; Operating System Kernel; Separation Logic;Rust; Theorem Proving

操作系统是计算系统的基础平台,操作系统安全性是现代操作系统中至关重要的一个方面.它不仅保护用户数据免受未经授权的访问和恶意软件的威胁,还确保系统的稳定性和可靠性.此外,安全的操作系统还能减少系统崩溃和性能问题,提高整体生产效率和用户信任度.因此,操作系统的安全性是维持数字生态系统健康和安

*基金项目: 国家自然科学基金(U2341212)

收稿时间: 2024-08-25; 修改时间: 2024-10-14; 采用时间: 2024-11-26; jos 在线出版时间: 2024-12-10

全的基石.

内核安全是操作系统安全的核心部分,因为内核负责管理系统的所有关键操作,包括内存管理、进程调度、硬件交互等.强化内核安全性可以有效防止恶意软件和攻击者获取核心权限,从而控制整个系统.

当前,Rust 是一种以内存安全和线程安全为核心设计原则的编程语言.它通过所有权机制、借用检查和生命周期管理来防止常见的内存错误,如空指针引用和数据竞争.这使得 Rust 在系统级编程中尤其受欢迎,因为它能够在编译时捕获许多潜在的错误,从而减少运行时崩溃和漏洞的可能性.然而,Rust 中也存在一个名为 unsafe 的关键字,它允许开发者绕过编译器的安全检查,以进行一些低级操作,如直接操作内存指针.虽然 unsafe 代码块在某些情况下是必要的,但它也引入了潜在的风险.如果使用不当,unsafe 代码块可能导致内存泄漏,数据竞争和其他安全漏洞.因此,对 unsafe 代码进行形式化验证是确保其安全性的重要手段.这种验证可以帮助确认 unsafe 块在特定条件下是安全的,从而增强整个系统的安全性和可靠性.

目前,许多院校机构都在尝试将 Rust 语言应用于操作系统内核的开发中,形式化方法[1]作为确保系统安全性和可靠性的有力手段,广泛应用于操作系统内核的安全性检验中,但针对 Rust 构建的操作系统内核的验证工作仍然缺乏,针对 Rust 特点的内核验证方法也处于探索阶段.

某内核是国产自主开发的 Rust 内核,采用平衡于用户态服务模式和内核态函数函数模式之间的内核态服务模式,在性能和可靠性上进行折衷,利用 Rust 的内存安全特性产生的原生的安全隔离,异步并发等减少性能的损失.可信的应用可以在用户态独占资源,较大提高性能的释放.

本文以某内核作为验证目标,提出了 GhostFunc 的 safe 代码块和 unsafe 代码块组合验证思路,后者的验证基于 λ_{Rust} 进行接口的构建与拓展.所有工作在 Coq 定理证明器中以 Iris 并发分离逻辑框架呈现.论文主要工作和创新如下.

(1)提出了 GhostFunc 的 Rust 系统组合验证思路.

(2)针对操作系统内核的验证拓展了 λ_{Rust} 语义模型.

(3)对 Rust 构建的内核的形式化模型提供了实例.

本文第 1 节主要介绍操作系统内核验证、Rust 形式化验证等的相关工作,第 2 节介绍 Coq 交互式定理证明器,Iris 分离逻辑框架和 λ_{Rust} 语义模型,并简要介绍某内核的设计结构.第 3 节介绍 unsafe 代码块语法和语义模型与建模.第 4 节讨论 GhostFunc 的构建与形式化.第 5 节介绍 safe 代码块的形式化以及整体的验证方案与流程.

1 相关工作

当前国内外针对操作系统内核的验证工作有很多,主要分为模型层面的验证和代码层面的验证.下面对这些工作进行介绍.

Feng 等人提出将 Rely-Guarantee 方法应用于验证并发程序变换,这一方法有效地处理了并发程序中的复杂性[2].随后, Fu 等进一步提出了一个具有多级中断的抢占式操作系统内核的实用验证框架,并成功将其应用于 μC/OS-II 嵌入式操作系统的验证工作[3].这一验证框架通过形式化的方法确保了操作系统内核在处理多级中断时的正确性和可靠性.Qiao 等组成的 SpaceOS 验证团队针对 SpaceOS 嵌入式实时操作系统[4]进行了一系列验证工作.他们使用 Event-B 抽象方案对内存管理模块进行了详细的验证工作[5].Gerwin Klein 等人对 seL4 内核的验证工作主要确认了内核的实现与其形式化规格完全一致.这一验证覆盖了内核的所有功能,包括线程调度、内存管理和进程通信等.通过使用定理证明器 Isabelle/HOL,他们成功地证明了 seL4 内核在各个方面的正确性,这一工作被认为是操作系统内核验证领域的一项具有代表性的成果[6-7].Andronick 等人对实时操作系统 eChronos 进行了验证[8],特别关注在多程序复杂并发以及多级嵌套中断场景下的内核调度性质.他们在 Isabelle/HOL 中对内核的同步和异步中断机制以及上下文切换调度栈进行了详细的建模,并通过证明调度栈在运行过程中保持一些基本属性,如线程调度栈的最底层保存的一定是当前线程等,得出调度属性在内核全局都保持的结论.这一验证确保了任何时刻运行的程序都是优先级最高的程序.Singularity 项目[9]

是由 Microsoft Research 发起的,旨在开发一种高可靠性的操作系统.在该项目中,研究人员使用了 Spec#和 Boogie 等验证工具对操作系统的关键组件进行了形式化验证,确保了组件之间的接口契约.这种契约式的验证方法提高了系统的可靠性,并显著减少了运行时错误的发生.

下面介绍 Rust 程序的形式化验证方面已有的一部分工作.

Cui 等人通过静态数据流分析检测 Rust 程序中的内存释放错误,提高了内存安全性[10].Dai 等人通过 Coq 定理证明器证明了 Rust 实现的页表在 Software Enclave Hypervisor 中的正确性[11]. CRust 采用有界模型检测的技术针对 Rust 中的不安全代码的内存安全性质进行验证[12].主要的技术原理是使用工具将 Rust 程序翻译为 C 语言实现的程序,然后在 CBMC 工具[13]中进行验证.Prusti[14-15]证明工具与 CRust 工具类似,不同的是先使用 MIR 提供的语言类型通过分离逻辑方式合成程序证明,然后将 Rust 语言转换为一种中间语言 Silver,并使用验证框架 Viper 中的符号执行工具进行验证.该工作的不足之处是不支持闭包,生命周期等特性的验证.RustBelt 是由来自 MPI-SWS 的 Ralf Jung 主导的一个研究项目[16-18],旨在对 Rust 语言的核心安全性进行形式化验证.RustBelt 使用了交互式定理证明器 Coq,通过 Iris 并发分离逻辑框架对 Rust 的类型系统和借用检查器进行了详细建模和验证.通过这项工作,研究人员证明了 Rust 的核心语言特性,如所有权和借用,能够确保内存安全性和数据竞争的防止.本文对于 unsafe 代码的语义模型的建立参考了该项工作.

当前关于操作系统内核的基于定理证明器的验证中针对 C/C++构建的传统架构内核居多,对于使用 Rust 构建的新型架构内核验证的相关工作相对空白,而针对 Rust 程序的验证,特别是基于定理证明的形式化验证工作非常少,RustBelt 项目虽然对 Rust 底层的安全机制进行了探索与验证,但并没有应用到实际 Rust 构建的系统中.

2 基础知识

本文在交互式定理证明器 Coq 中利用 Iris 验证框架对某使用 Rust 构建的新型架构内核中任务管理和调度模块的正确性进行验证.下面对基本知识进行简要介绍.

2.1 定理证明器Coq

Coq 是一个高级的交互式定理证明器[19],它允许用户编写形式化证明,这些证明可以由计算机检查以确保它们的正确性.Coq 使用了一种名为“构造演算”的形式语言,这是一个包含高阶逻辑的强类型函数式编程语言.它广泛用于研究领域,特别是在软件和硬件验证、形式化数学以及其他需要高可信度的场景中.Coq 的类型系统支持 dependent types,这意味着类型可以依赖于值.这种能力极大地增强了语言的表达能力,允许用户表达丰富的逻辑和数据结构.Coq 基于构造主义逻辑,这意味着它不接受经典逻辑中的排中律(每个命题都是真或假的)和选择公理.在 Coq 中,证明某事存在相当于提供一个构造该事物的方法.Coq 包含一个证明助手,帮助用户通过一步一步的 tactics 来构建证明.每个策略可以修改当前的证明状态或引入新的假设.Coq 支持模块化编程,允许用户编写可重用的库和模块,这些库和模块可以被不同的项目共享和使用.Coq 被用来验证软件程序的正确性,包括操作系统内核、编译器和其他关键系统.例如,CompCert C 编译器[20]就是一个被证明为语义保持的编译器,采用 Coq 进行正确性验证.Coq 也被用于形式化数学证明,包括著名的四色定理[21]和费马大定理[22]的部分证明.

2.2 并发分离逻辑证明框架Iris

Iris[23-25]是用于构建高级并发程序验证的框架,它在 Coq 定理证明器上实现.Iris 使用高阶分离逻辑,提供强大的抽象和模块化机制,这使得它非常适合用来验证复杂的并发系统和软件.Iris 基于高阶分离逻辑,能够处理可变状态,允许逻辑公式中包含资源的概念.分离逻辑特别适合于并发和并行计算的情境,因为它可以清晰地描述和推理那些由多个线程或进程共享和修改的资源.Iris 的计支持高度模块化,使得用户可以构建可复用的验证组件.这对于处理大规模软件系统特别重要,因为它减少了验证工作的重复性,并提高了代码的可维护性.Iris 能够处理动态分配的资源,用户可以在逻辑中动态地创建和销毁资源,这对于处理复杂的动态行为至关重要.IPM(Iris Proof Mode)提供了一组定制的策略(tactics),这些策略专门针对 Iris 逻辑的特点进行了优化.例如,

这里给出一个简单的并发加法器在 IPM 中的证明过程.首先定义加 1 和不变式以及需要证明的引理.

```
Definition incr (ℓ : loc) : expr := #ℓ <- !#ℓ + #1.
Definition incr_inv (ℓ : loc) (n : Z) : iProp := (∃ (m : Z), ⌜ (n ≤ m)%Z ⌝ ∗  ℓ ↦  #m)%I.
Lemma parallel_incr_spec (ℓ : loc) (n : Z):
      {{{ ℓ ↦  #n }}} (incr ℓ) ||| (incr ℓ) ;; !#ℓ {{{m, RET #m; ⌜ (n ≤ m)%Z ⌝ }}}.
```

不变量的主体定义为 counter_inv.这里使用%I 告诉 Coq 将逻辑公式解析为 Iris 断言,这意味着它将把连接词解释为 Iris 连接词.使用#来将 Coq 命题如变量,数字等嵌入到 Iris.在此示例中,将“≤”作为 Iris 断言.简单来说,这种嵌入意味着嵌入的断言要么适用于所有资源,要么不适用.⌜ ... ⌝用于将 Coq 断言嵌入到 Iris 断言.现在定义简单并行加法器的证明过程.

```
iIntros (Φ) "Hpt HΦ".
iMod (inv_alloc N _ (incr_inv ℓ n) with "[Hpt]") as "#HInv".
```

这段代码第一行介绍了证明中使用的假设和后续要证明的结论.Φ是后续要证明的结论,Hpt 是前置条件.第二行建立了一个名为 N 的不变量,保护位置 ℓ 的内容.这是为了确保在并发环境中,不同线程看到的 ℓ 的状态是一致的.后续证明都是使用类似的 Iris 证明策略.

2.3 lambdaRust

λ_{Rust} 是一种基于 Iris 逻辑框架开发的 Rust 形式化语义模型[22],专门用于验证 Rust 程序的内存安全性和并发正确性.通过利用 Iris 的分离逻辑和幽灵状态机制, λ_{Rust} 能够精确表达和推理 Rust 的所有权和借用规则,确保在多线程环境下的内存安全和数据竞争防护,从而在理论上严格验证 Rust 程序的正确性和可靠性.

在 λ_{Rust} 中,资源模型用于描述程序的状态和资源,如内存单元、锁、信号量等.资源模型允许对这些资源进行形式化描述和验证,确保在并发环境下的正确性和安全性.幽灵状态是一种在逻辑中引入额外状态的机制,帮助进行形式化验证.幽灵状态不影响程序的实际运行,但在验证过程中用于推理和证明.

3 unsafe 代码语法和语义模型

Rust 中 unsafe 代码绕过了内存安全保证的检查,对于 unsafe 代码的验证,我们在验证时,实现层接近于 MIR,在语义模型的建立时,本文参考了 λ_{Rust} 的语法和语义模型定义,在语法的定义上,采用 EBNF,包含基本语法元素如变量声明,函数定义,控制结构,数据结构等,特别强调所有权和借用规则.其类型系统包括基本类型(如 bool、int)、复合类型(如数组、切片)、泛型、生命周期标注.我们在其基础上,拓展了与操作系统内核相关的一系列语法和语义.

3.1 语法

在语法的定义中,对 Rust 一些语法糖进行了去糖,所以在基础语法方面更接近于 MIR,但又对内核中一些常见且容易证明正确的行为包装成语法糖,这样可以较大程度上简化我们验证的过程,例如上下文切换等操作,被我们作为了语法的一部分.针对此 λ_{Rust} 语法定义做了拓展和修改,我们定义了 7 种表达式类型:上下文,调用链,变量,路径,操作符,函数体以及中断.

定义 3.1(**内核验证语法定义**).

$$C ::= k \mid u \mid runtext() \in Context$$

$$c ::= from \mid to \mid Path \mid call() \in Chain$$

$$v ::= false \mid true \mid z \mid funrec\ f(x)\ ret\ k := F \mid PID \mid Device \in Val$$

$$p ::= x \mid p.n \mid context.k \in Path$$

$$I ::= v \mid p \mid p_1 + p_2 \mid p_1 - p_2 \mid p_1 == p_2 \mid syscall() \mid new() \mid delete \mid *p \mid p_1 := p_2 \mid p_1 := n * p_2 \mid newP(PID) \mid killP(PID) \mid \in Instr$$

$F ::= let\ x = I\ in\ F \mid if\ k\ then\ F_1\ else\ F_2 \mid newlft\ F \mid endlft\ F \mid case*\ p\ of\ \overline{F} \mid jump() \mid call\ f(x)\ ret\ p \in Func$

$H ::= handleInterrupt(type)\ do\ F \in Handle\_\text{irq}$

这里 C 为 Context 上下文的对应语法,k,u 分别处于内核与用户态,runtext 为切换内核状态的包装函数.C 表示调用链,包括调用链的来源,去向以及路径和调用等.值 Val 只包含最基本的数据类型和内核中基本的变量:布尔值,整数 z,内存中的位置 ℓ ,函数 funrec,进程 id,设备名,这里并没有表示内存块号,而是选择将其用 ℓ 表示.路径 Path 用来指向复合类型,n 为偏移量,通过增加由 p 表达的指针 n 个内存单元来递增指针,context 用于管理上下文.Instr 指令包含常见的算术操作,内核常见的 syscall()为系统调用总控函数,new 和 delete 为内存的分配和释放,*p 等表示从内存中加载操作以及赋值到内存等,newP 和 killP 表示进程的创建和删除.函数体用于管理数据流和指令流,通过将指令串联来管理控制流,支持递归的调用,newlft 和 endlft 为控制生命周期的辅助指令也称为“幽灵”指令,这是因为它们并不进行任何的操作并且也不存在于上下文中,只存在于证明之中.call f(x) ret p 为递归调用的绑定.x 是参数的绑定器列表,k 是返回延续的绑定器,函数可以通过 call f(x) ret k 进行调用,其中 x 是参数列表,k 是函数返回时应调用的延续,以此实现循环.中断处理是专为内核验证定义的语法类型,以处理内核中复杂的中断时的处理时无需将中断分解,以原子形式进行处理,之所以这么做是由于某内核中中断涉及到比较复杂的中断嵌套等情况,这样做可以简化验证,更加聚焦于本方案的模块验证中,但中断本身的验证仍需要进行,在本文中不详细讨论.

3.2 操作语义

在 λ_{Rust} 项目中定义了 λ_{Rust} 核心语言来具体描述 Rust 的形式.核心语言是一种具有指针运算和并发操作的 λ 演算,我们在文中也采用类似的方式来定义但加入了操作系统中如上下文切换、系统调用等的操作,选择 core 语言的目的是将内核中一系列 unsafe 的操作更便捷的建模,使推理更加简洁和清晰.core 语言模式如下.

$z \in \mathbb{Z}$

$Expr \in e ::= v \mid x \mid e_1.e_2 \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 \le e_2 \mid e_1 == e_2 \mid e(\overline{e}) \mid *^o e$

$\mid e_1 :=_o e_2 \mid CAS(e_0,e_1,e_2) \mid alloc(e) \mid free(e_1,e_2) \mid case\ e\ of\ \overline{e} \mid fork\{e\}$

$Val \in v ::= \varnothing \mid \ell \mid z \mid recf(\overline{x}) := e$

$Order \in o ::= sc \mid na \mid na'$

$LockSt \in \pi ::= writing \mid reading\ n$

$Ctx \ni K ::= K.e \mid v.K \mid K + e \mid v + K \mid K - e \mid v - K \mid K \le e \mid v \le K \mid K == e \mid v == K$

$\mid K(\overline{e}) \mid v(\overline{K} + +[K] + +\overline{e}) \mid *^o K \mid K :=_o e \mid v :=_o K \mid CAS(K,e_1,e_2) \mid$

$CAS(v_0,K,e_2) \mid CAS(v_0,v_1,K) \mid alloc(K) \mid free(K,e_2) \mid free(e_1,K) \mid case\ K\ of\ \overline{e}$

这里由于篇幅原因,我们不对这里的细节做过多的介绍.为了减少操作语义定义的工作量,我们的操作语义的定义也是基于 core 语言.

下面正式介绍我们基于 core 语言定义的操作语义,但是注意我们在此处并没有用到在 λ_{Rust} 中操作语义中所有的语言构体,而是选择了一部分,并在部分语义中做了构造的适应于内核验证的操作语义拓展,首先我们定义求值评估操作语义.

定义 3.2(**求值评估操作语义**).

$$\frac{}{\langle z,\sigma\rangle \to \langle z,\sigma\rangle}\ [\text{E-Z}] \qquad \frac{}{\langle x,\sigma\rangle \to \langle \sigma(x),\sigma\rangle}\ [\text{E-VAR}]$$

$$\frac{\langle e_1,\sigma\rangle \to \langle e_{1'},\sigma'\rangle}{\langle e_1 + e_2,\sigma\rangle \to \langle e_{1'} + e_2,\sigma'\rangle}\ [\text{E-ADD1}] \qquad \frac{\langle e_2,\sigma\rangle \to \langle e_{2'},\sigma'\rangle}{\langle v_1 + e_2,\sigma\rangle \to \langle v_1 + e_{2'},\sigma'\rangle}\ [\text{E-ADD2}]$$

$$\frac{v_3 = v_1 + v_2}{\langle v_1 + v_2,\sigma\rangle \to \langle v_3,\sigma\rangle}\ [\text{E-ADD3}]$$

$$\frac{}{\langle CAS(e_0,e_1,e_2),\sigma\rangle \to \langle v,\sigma'\rangle}\ [E-CAS] \qquad \frac{h \vdash v_1 = v_2}{\langle v_1 == v_2,h\rangle \to \langle 1,h\rangle}\ [E-EQ]$$

$$\frac{z_1 \le z_2}{\langle z_1 \le z_2,\sigma\rangle \to \langle 1,\sigma\rangle}\ [E-LE] \qquad \frac{z_1 > z_2}{\langle z_1 \le z_2,\sigma\rangle \to \langle 0,\sigma\rangle}\ [E-GT]$$

$$\frac{}{\langle alloc(v),\sigma\rangle \to \langle p,\sigma[p \mapsto v]\rangle}\ [\text{E-ALLOC}] \qquad \frac{}{\langle free(p),\sigma\rangle \to \langle unit,\sigma - \{p\}\rangle}\ [\text{E-FREE}]$$

$$\frac{}{\langle read(p),\sigma\rangle \to \langle \sigma(p),\sigma\rangle}\ [\text{E-READ}] \qquad \frac{}{\langle store(p,v),\sigma\rangle \to \langle unit,\sigma[p \mapsto v]\rangle}\ [\text{E-STORE}]$$

介绍操作语义之前,我们先解释操作语义中符号的含义.<>角括号用于表示一个有序对,其中第一个元素表示一个表达式(或操作),第二个元素表示一个状态(如内存状态).在操作语义中,这种表示方式用来描述程序在某个状态下执行某个表达式的过程,例如 $\langle expr,\sigma\rangle$ 表示在内存状态 σ 下执行操作 $expr$. σ 表示内存状态,通常是内存地址到值的映射. p 为内存地址即指针. $\sigma[p \mapsto v]$ 表示更新内存状态,将新分配的内存地址 p 映射到值. $\to$ 表示状态转变. $p \in dom(\sigma)$ 表示地址 p 存在于内存状态 σ 的域中. $unit$ 表示释放操作的返回值,通常为单位类型(无意义值), $\sigma - \{p\}$ 表示从内存状态 σ 中移除地址 p . $h \vdash v_1 = v_2$ 表示在内存状态 h 下, v_1 等于 v_2 .在判断操作语义中 1 表示 $true$,0 表示 $false$.

E-Z 表示常量 z 在任何状态 σ 下都保持不变,即不引起状态的改变,E-VAR 表示为变量 x 添加内存映射.E-ADD1 表示当 e_2 在状态 σ 下化简为 e_2' 且状态变为 σ' 时,表达式 $e_1 + e_2$ 化简为 $e_1' + e_2$,状态也变为 σ' .E-ADD2 跟 E-ADD1 类似,E-ADD3 表示当两个值 v_1 和 v_2 相加时,结果为 v_3 ,且状态 σ 保持不变.E-ALLOC,E-FREE,E-LOAD,E-STORE 四个操作是有关内存方面的操作.E-ALLOC 表示当在内存状态 σ 下分配一个值 v 时,如果 p 是一个新地址,则结果状态为新地址 p 和更新后的内存状态 $\sigma[p \mapsto v]$ .E-FREE 表示当在内存状态 σ 下释放地址 p 时,如果 p 存在于 σ 中,则结果状态为单位类型和更新后的内存状态(移除了 p).E-LOAD 表示当在内存状态 σ 下从地址 p 读取值时,如果 p 存在于 σ 中,则结果状态为读取的值 $\sigma(p)$ 和未改变的内存状态 σ .E-STORE 表示当在内存状态 σ 下将值 v 写入地址 p 时,如果 p 存在于 σ 中,则结果状态为单位类型和更新后的内存状态 $\sigma[p \mapsto v]$.

下面介绍求值评估操作语义之外的拓展操作语义部分.

定义 3.3 (**拓展操作语义**).

$$\frac{\langle e,\sigma\rangle \to \langle e',\sigma'\rangle}{\langle if\ e\ then\ s_1\ else\ s_2,\sigma\rangle \to \langle if\ e'\ then\ s_1\ else\ s_2,\sigma'\rangle}\ [O\text{-IF}]$$

$$\frac{\langle e,\sigma\rangle \to \langle true,\sigma'\rangle}{\langle if\ true\ then\ s_1\ else\ s_2,\sigma\rangle \to \langle s_1,\sigma'\rangle}\ [O\text{-IF-TRUE}] \qquad \frac{\langle e,\sigma\rangle \to \langle false,\sigma'\rangle}{\langle if\ false\ then\ s_1\ else\ s_2,\sigma\rangle \to \langle s_2,\sigma'\rangle}\ [\text{O-IF-FALSE}]$$

$$\frac{\langle e,\sigma\rangle \to \langle e',\sigma'\rangle}{\langle while\ e\ do\ s,\sigma\rangle \to \langle if\ e\ then\ (s;\ while\ e\ do\ s)\ else\ skip,\sigma'\rangle}\ [\text{O-WHILE}]$$

$$\frac{}{\langle call\ f(\overline{x})\ ret\ k,\sigma\rangle \to \langle let\ cont\ k(\overline{x}) := F_1\ in\ F_2,\sigma\rangle}\ [\text{O-CALL}] \qquad \frac{}{\langle ret\ v,\sigma\rangle \to \langle v,\sigma\rangle}\ [\text{O-RET}]$$

$$\frac{}{\langle fork\ \{e\},\sigma\rangle \to \langle e,\sigma\rangle|\langle e,\sigma\rangle}\ [\text{O-FORK}] \qquad \frac{}{\langle lock(p),\sigma\rangle \to \langle unit,\sigma[p \mapsto locked]\rangle}\ [\text{O-LOCK}]$$

$$\frac{h(\ell) = (reading\ 0,v')}{\langle CAS(\ell,v_1,v_2),h\rangle \to \langle 1,h[\ell \leftarrow (reading\ 0,v_2)]\rangle}\ [\text{O-CAS-SUCCESS}]$$

$$\frac{h(\ell) = (reading\ 0,v') * v' \neq v_1}{\langle CAS(\ell,v_1,v_2),h\rangle \to \langle 0,h\rangle}\ [\text{O-CAS-FAIL}]$$

O-IF 当条件表达式 e 在状态 σ 下简化为 e' 时,条件语句 $if\ e\ then\ s_1\ else\ s_2$ 简化为 $if\ e'\ then\ s_1\ else\ s_2$,状态也

变为 e'.O-IF-TRUE 表示在当条件表达式 e 在状态 σ 下化简为 true 时,条件语句 $if\ true\ then\ s_1\ else\ s_2$ 化简为 s_1 同时状态变为 e'.O-IF-FALSE 和 O-WHILE 与 O-IF-TRUE 的条件判断类似.

$call\ f(\overline{x})\ ret\ k$ 表示调用函数 f 并返回给 k .[O-CALL]表示当调用函数 f 时,将函数体 F_1 和 F_2 放入上下文中,并在内存状态 σ 下继续执行.[O-RET]表示当执行返回操作时,返回值 v 并且内存状态保持不变.[O-FORK]表示当创建一个新的并发线程时,结果状态包含两个线程各自执行表达式 e 的状态,而内存状态 σ 被共享.

CAS 操作在并发环境下是原子的,所以要确保 CAS 操作不能被打断.我们定义 $CAS(\ell,v_1,v_2)$ 表示比较并交换操作,将地址 ℓ 上的值从 v_1 更新为 v_2 , $h(\ell)=(reading 0,v')$ 表示在内存状态 h 中,地址 ℓ 的当前值为 v' 并且没有读写锁. $h[\ell\leftarrow(reading 0,v_2)]$ 表示更新内存状态,将地址 ℓ 的值设置为 v_2 .[O-CAS]表示当执行比较并交换操作时,如果地址 ℓ 上的值为 v_1 ,则将其更新为 v_2 并返回 1 表示成功;如果地址 ℓ 上的值不为 v_1 ,则返回 0 表示失败,内存状态保持不变.

4 任务管理与调度验证实例

本章将以任务管理模块验证为例,以第三章和第四章为基础形式化任务管理模块中的 unsafe 代码部分,并同时对 safe 代码部分进行抽象规约.建立两类形式化之间的接口,接口的定义会在 4.3 小节详细阐述.本节整体的验证方案如图 1 所示.

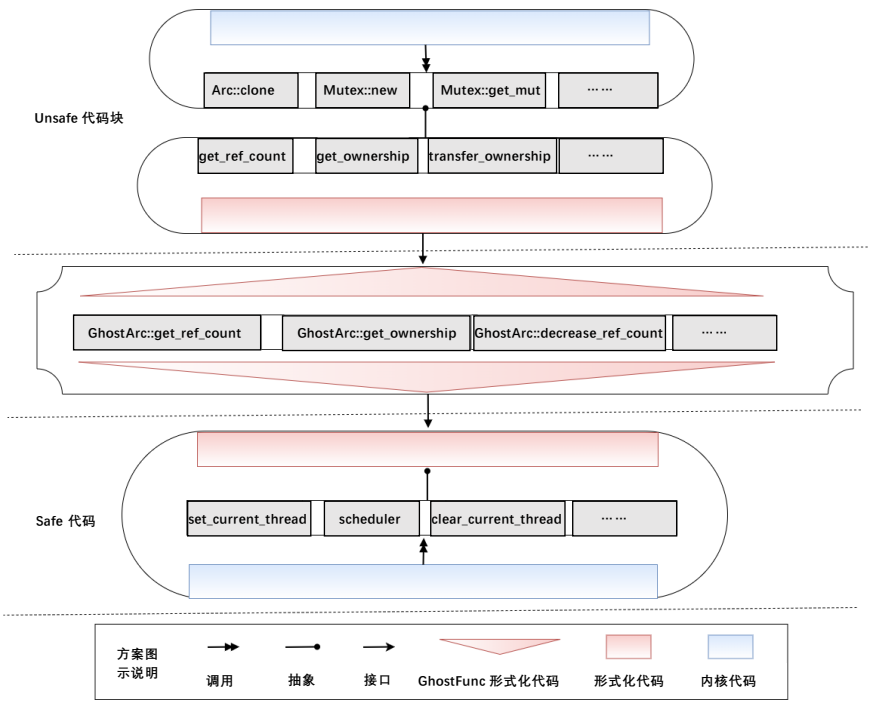

图 1 整体验证方案

GhostFunc 为 safe 代码块和 unsafe 代码块的验证代码提供了连接的接口,使得验证更具模块化.在保证验证细粒度的同时也降低了验证的复杂性.验证过程细节会在本章后续小节详细介绍.

4.1 任务管理unsafe代码段形式化

本节将对内核中 unsafe 代码段进行形式化建模.任务管理与调度模块代码(包含数据结构以及标准库)共 473 行代码.其中代码来源如下

表 1 模块代码统计

代码来源	行数

alloc::sync::Arc	91
alloc::sync::Weak	11
spin::sync::Mutex	46
spin::sync::rwlock	9
core::sync::atomic	4
core::ptr::read	49
core::ptr::write	38
core::slice	17
arch64(非标准库)	>100

本文对于 arch64 中的有关硬件的 unsafe 操作,没有进行形式化建模,所以本章的建模主要是针对标准库中的 unsafe 操作.下面以涉及代码量较多 Arc 和 Mutex 为例,对 unsafe 代码的形式化过程详细地介绍,主要针对其具体功能以及涉及到的如生命周期等属性进行建模和不变式的定义.

4.1.1 alloc::sync::Arc

Arc<T>类型提供了在堆中分配的 T 类型值的共享所有权,在 Arc 中调用 clone 方法会生成新的实例,实例指向堆上源 Arc 所指处,并增加引用计数.在某内核中,Arc 的使用非常频繁,主要是用于进线程以及权能相关的管理.Arc 的核心机制是引用计数,它通过原子计数器来跟踪对象有多少个活跃的引用.下面为 Arc 的三个基本定义.

定义 4.1(Arc 的基本属性定义).

$$\begin{aligned}
&Definition\ [\![Arc(\tau)]\!].size := [\![\tau]\!].size + 2.\\
&Definition\ [\![Arc(\tau)]\!].own(tid, vl) :=\\
&\qquad \exists rc\ l, vl = [\#l] *\\
&\qquad l \mapsto 1(rc, [\![\tau]\!].own\ tid) *\\
&\qquad rc \mapsto \{1\}\#1 *\\
&\qquad \square(rc \mapsto \{1\}\#1 \Rightarrow [\![\tau]\!].inv\ tid\ l).\\
&Definition\ Arc(\tau).shr(\kappa, tid, l) := \exists rc, rc \mapsto \kappa\#1 * [\![\tau]\!].shr\ \kappa\ tid\ l.
\end{aligned}$$

分别是 size,所有权 own,和共享属性,在不考虑 DST 的情况下,在类型系统中 $[\![\tau]\!]$ 和 $own(\tau)$ 在语义上被归为同一类,所以 $[\![\tau]\!]$ 的 size 属性与 $own(\tau)$ 相等,在 λ_{Rust} 中的类型如布尔类型,整形类型,和类型,积类型等的 size 属性都为 1,大于 1 的为复合或嵌套类型,例如 Arc 需要有引用计数,在类型上可视为 lock 大类中的需要额外属性的类型,所以在其 size 基础上加 2.

接下来介绍 Arc 的包含的三种方法的形式化定义:分别是新建 new(),克隆 clone(),删除 drop() .

定义 4.2(Arc 包含的方法形式化定义).

$$\begin{aligned}
&\{[\![\tau]\!].own(\overline{w})\}\ let\ x = Arc :: new(\overline{w}) := fn(\tau) \to Arc(\tau)\\
&\{[\![Arc\ \tau]\!].own([x])\}\ let\ y = x.clone() := fn(\&\alpha\ Arc(\tau)) \to Arc(\tau)\\
&\{[\![Arc\ \tau]\!].own([\overline{e}])\}\ let\ y = x.drop(\overline{e}) := fn(Arc(\tau)) \to ()
\end{aligned}$$

这里()表示空,&表示获取原 Arc 指针的复制.这三个方法对应的后置结果形式如下:

$$\begin{aligned}
&\exists \ell.x = [\ell] * \ell \mapsto [1] + +\overline{w} * shared_own(int, \ell, \overline{w}, 1, 1)\\
&[\![Arc\ \tau]\!].own([x]) * [\![Arc\ \tau]\!].own([y])\\
&[\![Arc\ \tau]\!].()
\end{aligned}$$

new 对 ℓ 地址处添加了一个拥有共享所有权的变量,对应于 Arc 的在 Rust 中的含义,clone 将共享所有权拆分为了两个,为了简化模型,没有采用 λ_{Rust} 中的所有权的 token 形式.

除了定义和操作的形式化,我们还需要为 Arc 定义不变式,首先我们分析 Arc 在内核中所需保持的不变性质,Arc 在内核中主要是用于维护全局的线程队列,所以第一个不变性质就是队列的在内存中的位置 ℓ 需要保持,第二是引用的计数正确,第三是正确的销毁.基于此我们可以定义出 Arc 的不变式 Arc_inv.

定义 4.3(Arc 不变式定义).

$$Arc\_inv_{\tau}(\kappa',\ell,t) := \ell \mapsto true \vee \ell \mapsto false * \kappa'_{full}(\exists \bar{v}.(\ell+1) \mapsto \bar{v} * [\![\tau]\!].own(t,\bar{v})) \vee \ell \mapsto false * t \mapsto ()$$

这里 κ' 用于表示生命周期的快照,t 为线程集合,第二个不变式状态表示若 Arc 处于空闲状态,可以使得其储存的下一个位置储存值对应于当前的线程,使其获得该复制位置的所有权.

4.1.2 alloc::sync::Mutex

标准库中的互斥锁.互斥锁阻止等待锁可用的线程.互斥锁可以通过 new 构造函数创建. 每个互斥锁都有一个类型参数,表示它正在保护的数据.只能通过从 lock 和 try_lock 返回的 RAII 保护来访问数据,这保证了只有在互斥锁被锁定时才可以访问数据.在 Arc 中我们没有详细介绍 Rust 中锁的机制,在这里将进行介绍.锁保护的内容的所有权应当拥有一个完整的借用,若没有获取到完整的借用,那他的外层也就是锁本身应当包含这个内容的借用,在形式上我们将这两种情况统一起来,如果 κ' 在锁由某个线程持有的情况下结束,那么这个线程就会因此拥有这个锁原本含有的内容.这就是为什么我们在介绍 Arc 的时候 κ' 的下标为 full,也就是完整的借用.

下面介绍 Mutex 的基本定义.

定义 4.4(Mutex 的基本属性定义).

$$Definition\ [\![Mutex(\tau)]\!].size := [\![\tau]\!].size + 1.$$

$$Definition\ [\![\mathrm{Mutex}(\tau)]\!].own(t,\bar{v}) := [\![bool \times \tau]\!].own(t,\bar{v})$$

$$\begin{aligned} Definition\ & [\![Mutex(\tau)]\!].own(tid,vl) := \\ & \exists rc\ l, vl = [\#l] * \\ & l \mapsto \mathrm{l}(rc, [\![\tau]\!].own\ tid) * \\ & rc \mapsto \{1\}\#1 * \\ & \square(rc \mapsto \{1\}\#1 \Rightarrow [\![\tau]\!].inv\ tid\ l). \end{aligned}$$

$$Definition\ \ Mutex(\tau).shr(\kappa, tid, l) := \exists rc, rc \mapsto \kappa\#1 * [\![\tau]\!].shr\ \kappa\ tid\ l.$$

定义和前面 Arc 的定义一致,但由于在内核中我们不会对 Mutex 类型保护的数据进行共享,也不会涉及到内部可变性的运用,所以在此处我们不同于 λ_{Rust} 中的定义 shr 的属性控制其生命周期与堆叠借用等,而是将其与一个 bool 值相对应,锁的状态改变不涉及非原子操作.

下面介绍 Mutex 方法 new,lock,get_mut 方法定义.

定义 4.5(Mutex 包含的方法形式化定义).

$$[\![Mutex(\tau).new(w)]\!] := fn() \to Mutex(\tau)$$

$$[[\![Mutex(\tau).lock(l)]\!] := fn(k) \to \exists v, l \mapsto true * [\![\tau]\!].own(t,v)$$

$$[[\![Mutex(\tau).get\_mut(l)]\!] := fn() \to \exists v, l \mapsto true * [\![\tau]\!].own(\_)$$

由于 Mutex 的形式化定义相较于 Arc 的结构简单一些,所以此处没有采用 Arc 的前后置条件的展示方法,这里操作的返回值即是后置条件.new 方法用于创建新 Mutex 实例,并初始化内部的数据,返回新的实例.lock 方法用于尝试获取 Mutex 的锁,并返回保护数据的引用,这里表示尝试获取锁 l,如果获取成功,则返回一个线程 t 以及被保护的数据 v 的所有权.get_mut 用于在不需要获取锁的情况下,获取 Mutex 内部数据的可变引用,涉及 Rust 中的可变引用机制,在内核中用处主要用于内核模块加载过程的初始数据结构的加载等,因为在这个阶段,尚未有其他线程运行,因此可以安全地使用 get_mut 方法对数据结构进行初始化.

我们分析 Mutex 在内核中所需保持的不变性.Mutex 在内核中主要用于确保并发访问的安全性,保证在任何时刻只有一个线程可以访问被保护的数据.因此,第一个不变性就是被保护的数据的访问权限需要保持一致.其次,Mutex 需要确保其锁的状态在任何时刻都准确反映当前的持有状态.基于此,我们可以定义出 Mutex 的不变式 Mutex_inv.

定义 4.6(Mutex 不变式定义).

$$Mutex\_inv_{\tau}(l,t,v) := l \mapsto true * (t \mapsto own(t,v)) \vee l \mapsto false$$

这里,l 表示 Mutex 的锁状态,t 表示线程,v 表示被保护的数据.当锁被持有时,锁的状态 l 为 true,且存在一个线程 t 持有 Mutex,并拥有被保护数据 v 的所有权;当锁未被持有时,锁的状态 l 为 false.

4.2 GhostFunc形式化规范

在上一章,我们完成了对于某内核中 unsafe 代码的形式化工作,定义了一系列语义语法,在本章我们将对 unsafe 代码与 safe 代码之间的接口定义形式化规范.我们提出一种命名为 GhostFunc 的接口定义方法,也就是为 unsafe 代码定义辅助的接口“函数”,在 GhostFunc 中完成一次抽象规约,规约后的接口可以与 safe 代码进行组合验证.

4.2.1 GhostFunc 定义

GhostFunc 取名来自于 GhostState,用于辅助两个抽象层级不同的模块进行组合验证.以 Arc 的 GhostFunc 定义介绍在图 2 中介绍辅助验证流程,Unsafe Code Module 和 Safe Code Module 通过 GhostFunc 进行组合,从而完整整个系统的验证.

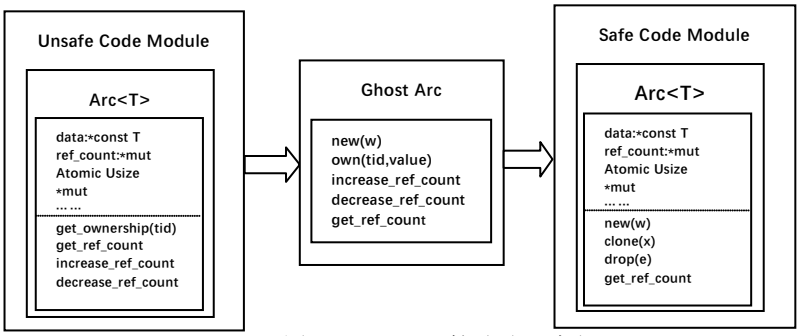

图 2 GhostFunc 辅助验证流程

为了显式表达模块间的接口,我们在 Unsafe Code Module 定义了的方法如下:

- get_ownership(tid):获取特定线程的所有权.
- transfer_ownership(tid_src,tid_dst):在源线程和目标线程之间转移所有权.
- get_ref_count(): 获取当前的引用计数.
- increase_ref_count(): 增加引用计数.
- decrease_ref_count(): 减少引用计数,包含减少到 0,即 drop().

下面具体介绍这些用于 GhostFunc 的接口函数在 Iris 中的形式化定义.

定义 4.7(GhostArc 形式化定义).

$$\forall \tau, v.[\![\tau]\!].own(v) -\!\!* |==> [\![\tau]\!].size \Leftrightarrow \exists l\ k, l \mapsto (k,[\![\tau]\!].own) * k \mapsto [\![\tau]\!].size. (1)$$

$$\forall \tau, v.[\![\tau]\!].size -\!\!* |==> [\![\tau]\!].shr(v) \Leftrightarrow \exists rc, rc \mapsto [\#1] * [\![\tau]\!].inv(v). (2)$$

$$\forall \tau, x.[\![GhostArc(\tau)]\!](get\_ref\_count(x)) -\!\!* |==> [\![\tau]\!].size \Leftrightarrow \exists rc, rc = x.ref\_count \wedge rc \mapsto 1 * [\![\tau]\!].shr(x). (3)$$

$$\forall \tau, x.[\![GhostArc(\tau)]\!](decrease\_ref\_count(x)) -\!\!* |==> [\![\tau]\!].size \Leftrightarrow \exists rc, rc = x.ref\_count \wedge rc \mapsto 1 * [\![\tau]\!].inv(x). (4)$$

$$\forall \tau, x, tid.[\![GhostArc(\tau)]\!](get\_ownership(x,tid)) -\!\!* |==> [\![\tau]\!].size \Leftrightarrow \exists rc\ l, rc = x.ref\_count \wedge l = x.data \wedge rc \mapsto 1 * [\![\tau]\!].inv(tid). (5)$$

$$\forall \tau, x, tid\_src, tid\_dst.[\![GhostArc(\tau)]\!](transfer\_ownership(x,tid\_src,tid\_dst)) -\!\!* |==> [\![\tau]\!].size \Leftrightarrow \exists rc\ l, rc = x.ref\_count \wedge l = x.data \wedge rc \mapsto 1 * [\![\tau]\!].inv(tid\_src) == * [\![\tau]\!].inv(tid\_dst). (6)$$

$$\forall \tau, x.[\![GhostArc(\tau)]\!](clone(x)) -\!\!* |==> [\![\tau]\!].size \Leftrightarrow \exists rc\ l, rc = x.ref\_count \wedge l = x.data \wedge rc \mapsto 1 * [\![\tau]\!].inv(x). (7)$$

$$\forall \tau, x.[\![GhostArc(\tau)]\!](drop(x)) -\!\!* |==> [\![\tau]\!].size \Leftrightarrow \exists rc\ l, rc = x.ref\_count \wedge l = x.data \wedge rc \mapsto 1 * [\![\tau]\!].inv(x) == *(rc \mapsto 0 == *l \mapsto \_). \quad (8)$$

在 GhostArc 中我们共定义了 6 个接口,分别为获取引用计数,增加引用计数,减少引用计数,转移所有权,克隆对象,删除对象.公式 1 和公式 2 描述了 own 所有权和 size 大小属性之间的关系以及 shr 共享属性和 size 之间的关系.公式 3-8 中描述了一系列的 GhostArc 接口函数, $-\!\!*|==>$ 表示在消耗拥有关系的前提下,能够推导出后置的状态;==*是 Iris 框架中表示更新的逻辑操作符; $\ell \mapsto (k, [\![\tau]\!]) * k \mapsto [\![\tau]\!].size$ 表示 ℓ 映射到 k 和所有权关系,并且 k 映射到大小属性.

在 Rust 的 Arc 中,实际是不存在转移所有权操作的,在此处的转移所有权是一类辅助操作,用于在内核中显示的表达内核对象的转移和所在上下文的变化.在 4.1.1 小节 Arc 的形式化中,我们主要介绍了 Arc 相关方法的形式化,在 GhostFunc 中我们仍然实现了这些方法的接口,但添加了有关获取所有权以及增减引用计数的接口,这是为了后续在与 safe 代码过程中,能够更好的与内嵌该数据结构的对象的进行交互和组合.

4.2.2 GhostFunc 状态与不变式

为了保证验证的模块化,除去全局的不变式性质,我们在每个 GhostFunc 中都定义了局部的不变式,主要是保证各个 GhostFunc 中都保证所有权,生命周期等性质.并且在验证出现问题时,可分离出来进行排查验证.我们在本节用例子简单介绍 GhostFunc 状态和部分不变式性质.

为了清晰描述每个 GhostFunc 操作,首先需要定义该 GhostFunc 系统的状态.这里还是以 GhostArc 的系统状态为例介绍.状态的定义包括如下引用计数,数据,所有权,锁的状态等.

对于每个 GhostFunc,内部都有部分的操作的接口,我们在 GhostFunc 中证明操作的正确性,因为在 Unsafe Code Module 中已经进行证明,但仍需定义操作的前后置条件,作用第一是为了更好衔接 Safe Code Module,第二是在 Coq 中的实际证明中,我们发现如果在上下文不完整的情况下,编译选项会变得十分的复杂和冗余,给验证造成较大的困扰.同时我们显式的定义 Arc 的操作为 GhostArcOp 来描述 GhostArc 对外的接口.

下面 Coq 代码是以 GhostArc 中 clone 操作的前后置条件等为例介绍.

```
Record State (T : Type) : Type := {
    ref_count : nat;
    data : T;
    ownership : gmap nat nat;
    is_locked : bool;
  }.
  ……

Inductive GhostArcOp (T : Type) : Type :=
  | Clone (s : State T) : GhostArcOp T
  | Drop (s : State T) : GhostArcOp T
  | New (s : State T) (tid_src tid_dst : nat) : GhostArcOp T.

  ……

Definition pre_clone (s : State T) : iProp Σ :=
    ⌜ s.ref_count > 0 ⌝.

Definition post_clone (s s' : State T) : iProp Σ :=
   ⌜ s'.ref_count = s.ref_count + 1 ∧
    s'.data = s.data ∧
    s'.ownership = s.ownership ∧
    s'.is_locked = s.is_locked ⌝.
```

其中 State 为我们定义的 GhostArc 状态:ref_count 为引用计数,data 为数据,ownership:gmap 为拥有数据的线程 id 到引用计数的映射.后面归纳定义三个对外操作的接口.定义的 pre_clone 和 post_clone 分别为 clone 操作的前后置条件,前置条件为当前状态 s 的引用计数 s.ref_count 大于 0,确保了对象可以被有效克隆,也保证了 safe 代码部分能够合理使用该 GhostFunc.后置条件,即被克隆后应满足应用计数增加,数据不变,所有权不变,锁定状态不变,实际上我们整个验证中,并未涉及所有权转移的操作.

之后,我们显式的定义 GhostArc 的接口.

```
Definition ghost_arc_clone (γ : gname) (s : State T) : iProp Σ :=
    ∃ s', ⌜ pre_clone s ⌝ ∗
          ⌜ post_clone s s' ⌝ ∗
          own γ (● (Excl' s')) ∗
        inv γ (valid_ref_count s' ∗ valid_ownership s' ∗ valid_locked s').
```

其中 γ : gname 是 GhostArc 对象的全局名称,用于标识具体的 Arc 实例.s : State T 是当前的状态.pre_clone s 是克隆操作的前置条件.post_clone s s' 是克隆操作的后置条件.own γ (● (Excl' s')) 表示当前线程拥有的全局状态.inv γ (valid_ref_count s' ∗ valid_ownership s' ∗ valid_locked s')表示全局状态的不变式.

4.3 safe代码段形式化

除去在 4.1 小节已经完成的,还有 safe 代码段需要形式化.本文不同于 CertiKOS 的分层验证思路,而是采用逐步建模形式化的方式,主要在于 Rust 的正常 safe 代码块已经满足了内存的安全性,从验证的角度出发,我们无需额外考虑内存安全的形式化验证,只需考虑代码逻辑方面的正确性.从而简化了验证的过程.在本小节中,我们一共建模 277 行 Rust 代码.涉及到 Definition,RA,以及 proof 的 Iris 代码共 1900 余行.在本节我们简要介绍某内核的任务管理和调度模块并简要介绍模块建模以及其与 GhostFunc 的接口的形式化叙述.

4.3.1 数据结构的建模验证

在任务管理与调度模块中,参与验证的数据结构为全局的线程集合,某内核在任务管理中维护这个对象:THREAD_DEQUE: Lazy<Mutex<VecDeque<Option<Arc<Thread>>>>>,其中 Arc:进程引用计数,一个线程可能在 THREAD_DEQUE 或者所属进程的线程队列中有多份引用,VecDeque:线程队列,可以在队列两端执行插入、删除操作.在当前内核版本中的调试运行中,线程队列暂时没有双端操作,功能与普通队列相同.结构如图 3 所示

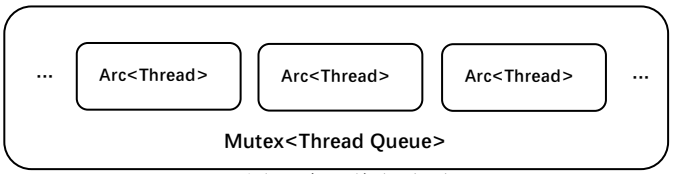

图 3 全局线程队列

在验证数据结构时,首先需要验证数据结构整体的正确性,所以需要对数据结构做简化,我们此处将队列中的 Arc<Thread>简化为 Lock<Thread, Key>,将队列中 Mutex<Thread>简化为 Lock<Thread Queue, Key>,表示将 Mutex 和 Arc 都简化为锁 Lock,但访问受保护的内容时,需要提供凭证 Key,这样可达到数据结构的基本功能,而 Arc 和 Mutex 本身的验证,我们会在 GhostFunc 中完成.基于此,我们在 Coq 中对该结构进行建模与验证.

关于 lock 的建模,主要涉及三个操作,分别是获取锁,释放锁和新建锁,并为锁定义了两个状态分别为 is_lock 和 locked,这些建模不详细解释,方法比较传统.下面介绍我们定义的不变式.

定义 4.8(全局队列不变式)

$$
\begin{aligned}
&lock\_inv(\gamma,\iota,\kappa,arr,c,ap,nextPtr,R) = \exists(o,i:nat)(xs:list\ bool),\\
&\begin{cases}
lengthxs = cap\\
\wedge nextPtr \mapsto \#(o+i)\\
\wedge is\_array(arr,xs)\\
\wedge invitation(\iota,i,cap)\\
\wedge own(\gamma,?(Excl'o,GSet(set\_seqoi)))\\
\wedge \begin{cases}
(own(\gamma,\circ(Excl'o,GSet(\varnothing))) \wedge R \wedge both(\kappa) \wedge xs = list\_with\_one(cap,(o\ mod\ cap)))\\
\vee(issued(\gamma,o) \wedge right(\kappa) \wedge xs = replicate(cap,false))\\
\vee(issued(\gamma,o) \wedge left(\kappa) \wedge xs = list\_with\_one(cap,(o\ mod\ cap)))
\end{cases}
\end{cases}
\end{aligned}
$$

o 和 i 是自然数,分别表示当前锁持有者的位置和等待队列中的线程数.xs 是一个布尔列表,表示锁的状态数组.arr 是锁状态队列的位置.nextPtr 是指向下一个 key 的位置.cap 是锁的容量,即锁状态数组的大小.R 是临界区资源.第一个算子表示锁的状态数组 xs 的长度必须等于锁的容量 cap,第二个算子表示 nextPtr 存储当前 key,即等待队列的长度,第三个算子表示锁的状态数组存储在位置 arr,并且当前状态为 xs.第四个算子表示当前锁持有的 key 数量为 i,总容量为 cap.第五个算子表示锁的拥有者是 o,当前的队列范围是 [o, o + i),通过 auth 机制确保其排他性.第五个算子表示锁的状态分为三种:开放状态,半开放状态和关闭状态,半开放状态锁的状态数组中所有位置均为 false.这个不变式确保了锁在各种操作(如获取和释放)中的正确性和一致性.可以看以下例子理解详细过程.:

假设锁的容量为 3,当前状态如下:

- o= 0,表示当前锁的持有者位置.
- i = 2,表示有两个线程在等待队列中.
- xs = [true, false, false],表示锁的状态数组.

在这种情况下,不变式 lock_inv 确保:

- 锁的状态数组长度为 3.
- key 值 nextPtr 存储的值为 0 + 2 = 2.
- 锁的状态数组 arr 存储当前状态 [true, false, false].
- 锁的拥有者是位置 0,当前的队列范围是 [0, 2).
- 锁的状态为开放状态,即 xs = list_with_one(3, 0).

4.3.2 任务管理与调度模块建模

在任务管理与调度模块,某内核主要涉及一个主循环,所有的调度等操作都存在于主循环中,调度器中包括三个函数:

set_current_thread:选择线程队列中的第一个 Ready 线程作为当前线程,无可用线程时进入待机等待中断

run_current_thread:循环调用 thread 的 run 函数执行当前线程,线程执行中遇到系统调用、中断、异常后 run 函数退出,再调用 handle_user_trap 函数处理陷入内核原因.完成处理后根据线程状态进行不同处理:

- Ready:处理完当前陷入线程可继续执行.
- Wait:线程执行异步等待系统调用陷入内核,在内核中被设为 Wait 状态.
- Suspended:线程运行时遇到时钟中断陷入内核,线程时间片用完时被设为 Suspended 状态.
- Exit:线程执行 ky_proc_exit 系统调用陷入内核,在内核中被设为 Exit 状态.

clear_current_thread:根据线程状态清除当前线程

- Suspended:说明线程已经经历了中止,将状态改回 Ready,再重新加入队尾.
- Wait:将在协程函数中将状态改回 Ready(后续章节介绍),再重新加入队尾.
- Exit:说明线程已终止,无需加入队尾,此时需要根据终止的线程是否为根线程,进行相应的清理操作.

在内核主循环的最后,循环依次调用上述三个函数.当线程队列为空时,系统进入待机.每次调用完三个函数后,执行一次协程调度器.

我们的建模是从代码层面上建模的,下面介绍线程状态清除的建模.其余建模与此类似.

对于 clear_current_thread,并且在仅考虑 GhostArc 的情况下,在执行前的状态确定如下,表示当前必须要有一个正在运行的线程.

$$pre\_clear\_current\_thread(s) := \exists t, s.current\_thread = Some(t)$$

在执行后状态确定为:

$$post\_clear\_current\_thread(s) := \exists t, s.current\_thread = Some(t) \wedge$$

(match t.state with

$|\ Suspended => s'.suspended\_queue = t :: s.suspended\_queue \wedge s'.current\_thread = None \wedge update\_ref\_count(s,s')$

$|\ Wait => s'.wait\_queue = t :: s.wait\_queue \wedge s'.current\_thread = None \wedge update\_ref\_count(s,s')$

$|\ Exit => s'.current\_thread = None \wedge remove\_ref\_count(s,s')$

$|\ Ready => s'.ready\_queue = t :: s.ready\_queue \wedge s'.current\_thread = None \wedge update\_ref\_count(s,s')$

$end)$

此处的有关引用计数的一系列操作,来自于我们之前在 GhostArc 中定义的一系列对外的接口,所以即使此处的建模抽象层级达不到控制引用计数的程度,我们依然可以通过 GhostArc 进行这种细粒度更高的操作.

下面介绍此验证如何与 GhostArc 组合验证,我们需要定义相关的接口,在定义接口时,我们着重维护 GhostArc 的引用计数以及所有权的属性等不变式.不变式的形式化定义如下:

$$\forall \gamma, \forall s, \exists s', pre\_clear\_current\_thread(s) \rightarrow post\_clear\_current\_thread(s,s') \wedge own(\gamma, Excl'(s')) \wedge inv(\gamma, valid\_ref\_count(s') \wedge valid\_ownership(s') \wedge valid\_locked(s'))$$

Arc<T>在 safe 代码接口在 Coq 中的形式化如下.

```
Definition update_ref_count (s s' : State T) : iProp Σ :=
  ∃ rc, rc ↦ s.ref_count ∗
  ⌜ s'.ref_count = s.ref_count + 1 ⌝ ∗
  rc ↦ s'.ref_count.

Definition remove_ref_count (s s' : State T) : iProp Σ :=
  ∃ rc, rc ↦ s.ref_count ∗
  ⌜ s'.ref_count = s.ref_count - 1 ⌝ ∗
  rc ↦ s'.ref_count.
```

接口的定义在满足前后置条件后继而保证整个大系统的一致性,即可组合完成.

4.3.3 模块内 API 验证

对于任务管理与调度模块,对外暴露的 API 包括创建线程(Create Thread API),调度线程(Thread Scheduler API),运行当前线程(Run Current Thread API),清除当前线程(Clear Current Thread API).

下面以运行当前线程为例介绍模块内 API 验证过程.

run_current_thread 函数的主要作用是从全局变量中获取当前线程并执行它,直到线程状态不再是 Ready. 我们需要验证以下关键点:

- 函数在当前线程为 Some(thread) 且状态为 Ready 时,正确地执行该线程.
- 在执行线程时,线程状态的变化符合预期.
- 函数在处理用户态陷入内核态的异常时,不会导致不一致的状态或数据破坏.

基于此,我们定义本 API 的不变式如下.

不变式 1:线程的状态一致性:

```
Definition thread_state_invariant (t : Thread) : iProp Σ :=
  ⌜ t.state = Ready ⌝ ∨
  ⌜ t.state = Running ⌝ ∨
  ⌜ t.state = Suspended ⌝ ∨
  ⌜ t.state = Exit ⌝.
```

不变式 2:系统中断状态一致性

```
Definition irq_state_invariant (s : State T) (t : Thread) : iProp Σ :=
  ⌜ t.state = Running -> s.irq_state = Disabled ⌝ ∧
  ⌜ t.state = Suspended -> s.irq_state = Enabled ⌝ ∧
  ⌜ t.state = Exit -> s.irq_state = Enabled ⌝.
```

不变式 3:用户态陷入内核态的异常处理一致性

```
Definition user_trap_invariant (t : Thread) (ctx : Context) : iProp Σ :=
  ⌜ valid_context(ctx) ⌝ ∧
  ⌜ handle_user_trap(t, ctx) = Some(new_ctx) ⌝ ∧
  ⌜ t.state = Running ⌝.
```

具体来说,这些不变式可以帮助我们验证:

线程状态的变化是合法的:线程在 Ready 状态时执行,并且仅在合法的状态序列中转换.

中断状态的设置是正确的:中断的启用和禁用与线程的状态保持一致,不会导致在执行过程中发生意外中断.

异常处理流程的完整性:在处理用户态陷入内核态的过程中,能够正确处理异常,并且在异常处理结束后,系统和线程状态保持一致.

下一步为细化细化前置条件和后置条件并进行形式化建模,建模过程中,需要将涉及到的 unsafe 代码块抽象出 GhostFunc 接口.例如在此 API 中涉及到的中断状态判断中大部分为 unsafe 代码,此处涉及约 150 行 Coq 代码,由于篇幅原因,展示其 GhostFunc 定义的一部分

```
Definition ghost_handle_irq (s : State T) : iProp Σ :=
  ∃ (irq_state_before irq_state_after : IrqState)
    (thread_before thread_after : Thread)
    (context_before context_after : Context)
    (ticks_before ticks_after : nat),

    (* 保存当前的中断状态 *)
    (⌜ irq_state_before = s.irq_state ⌝ ∗
     arch_save_irq_state(irq_state_before) ∗

(* 中断状态保存 *)
Definition arch_save_irq_state (irq_state : IrqState) : iProp Σ :=
    (* 假设我们有一个低级别的内存模型来表示中断状态的保存 *)
    ⌜ irq_state = Disabled ⌝ ∨ ⌜ irq_state = Enabled ⌝ ∗
    own (irq_state ↦ saved_state).

(* 中断状态恢复 *)
Definition arch_restore_irq_state (irq_state : IrqState) : iProp Σ :=
    (* 恢复之前保存的中断状态 *)
    saved_state ↦ irq_state ∗
    ⌜ irq_state = Disabled ⌝ ∨ ⌜ irq_state = Enabled ⌝.
```

在验证过程中即可结合ghost_handle_irq 的定义对 API 运行当前线程 run_current_thread 进行验证.

4.4 GhostFunc验证方法分析

本章是本文的重点章节,详细介绍了 GhostFunc 的形式化叙述,并以例子介绍了部分 unsafe 代码的建模以及在 GhostFunc 中的接口定义,同时介绍了 safe 代码部分以及任务管理与调度模块框架的形式化验证过程.

我们的形式化模型通过一系列精确的定义,捕捉了操作系统内核中关键操作的行为.例如,模型中定义的 run_current_thread 函数通过对线程状态和中断状态的不变式维护,准确反映了实际代码中的操作逻辑.具体来说,模型中的 ghost_handle_irq 函数抽象了代码中涉及的 unsafe 中断处理操作,并通过保持中断状态和线程状态的一致性,确保了代码在执行中的安全性.

这种模型与代码之间的紧密对应关系,表明我们的形式化模型在捕捉系统行为方面是准确且有效的.尽管没有完成全部的精化验证,我们可以通过模型与代码的结构和逻辑的相似性来推断它们之间的一致性.

在我们的形式化模型中,定义了一系列关键的不变式,如 thread_state_invariant 和 irq_state_invariant,这些不变式通过模型的操作函数得到了严格的维护.这些不变式不仅确保了模型中各个操作的正确性,同时也为代码实现提供了可靠的理论基础.

特别是通过对 GhostFunc 的引入,我们能够将 unsafe 代码中的复杂操作抽象出来,并通过安全接口与 safe 代码结合.这种方法确保了中断处理和线程状态管理在 safe 和 unsafe 代码之间的一致性,从而保证了系统整体行为的正确性.虽然尚未完成代码的精化验证,但基于模型中不变式的维护,我们可以推断代码实现能够在很大程度上保持与模型一致.在实际验证中,由于工作重点在于检验 GhostFunc 验证方案在实际工程中的有效性,我们在一些边缘情况验证和较复杂的情况做了一部分的简化假设.尽管我们的模型验证覆盖了大部分系统行为,但在某些情况下可能存在偏差.以下是我们识别出的潜在不一致性.由于在构建抽象模型时,我们对系统的一些行为进行了简化:

(1)忽略内存分配失败:在高层模型中,假设每次 alloc() 调用都能成功.然而,在实际实现中,可能会因为内存不足而导致分配失败,这种情况没有在模型中完全覆盖.未来工作中,我们计划扩展模型以捕捉内存不足的场景,并通过状态转移图模拟不同内存状态下的系统行为.

(2)中断与异常嵌套的简化:模型中对中断和异常的处理采用了顺序执行的假设,但在实际系统中,可能会发生中断嵌套或多级异常的情况.我们将通过引入状态机模型模拟中断优先级管理,并在实现代码的关键路径上进行不变量检查.这些情况可能在实际内核运行中产生验证结果和实际不一致的情况.

此外,为了测试一致性的情况,我们专门设计了一些测试场景如表 2,以测试模型和实际代码的一致性情况.

表 2 测试场景

TestCase	初始状态	输入条件	模型预期输出
TC-001	state = unlocked	调用 lock()	state = locked
TC-002	所有寄存器保存完成	任务切换指令	reg_saved = true
TC-003	mem_available = true	调用 alloc()	alloc_result = success
TC-004	mem_available =false	调用 alloc()	alloc_result = failure
TC-005	文件不存在	syscall_open("test")	syscall_result = error(404)
TC-006	当前处理中断级别为 2	嵌套中断	nested_interrupt_handled = true
TC-007	state = valid	GhostFunc 调用 unsafe 函数	state = valid
TC-008	input = -1	GhostFunc 输入非法值	error = invalid input

在表格中的各类场景设计中,预期输出和一致性的检查均通过,但是上述例如内存不足场景我们用变量来控制,但是实际中操作系统可能无法检测到实际的内存大小. 构造,其次是当中断嵌套到 3 级以上的时候,场景构建会会变得十分复杂,不易构建. 在二级中断中,我们顺序执行的中断处理方式仍然可以保持一致性.

本文提出的方法在设计时特别考虑了其可拓展性,以便能够适应操作系统和应用环境中不断变化和发展的需求.通过模块化的设计和抽象化的接口定义,我们的方法能够灵活地适应不同的系统配置和特性.例如,GhostFunc 模块的引入,使得复杂的 unsafe 操作可以被抽象为安全接口,并且这些接口可以根据具体需求进行扩展或替换,以支持特定的硬件中断处理或优化的线程调度算法.

此外,由于某内核正处于高速开发迭代中,当系统引入新功能或新特性(如多级缓存管理、虚拟化支持等)时,现有方法能够通过扩展接口或增加新的模块来进行无缝集成,而无需对核心模型进行大幅修改.这种设计不仅保证了当前系统的稳定性和安全性,还为未来的功能扩展提供了良好的基础,确保方法在面对新挑战时的适应性.

5 总 结

本文以某 Rust 构建的内核作为验证目标,提出了 GhostFunc 的 safe 代码块和 unsafe 代码块组合验证思路.重点介绍了适应于内核验证的 unsafe 代码形式化语法和语义模型、GhostFunc 的形式化规范.并以某内核任务管理与调度模块的验证作为实例进行分析,将 unsafe 和 safe 代码段从不同层级进行抽象,详细验证了任务管理与调度模块的安全属性包括:(1)无常见内存错误如缓冲区溢出、整型溢出、野指针;(2)代码的功能符合设计目标;(3)上下文的切换返回地址正确.

本文提出的方法在实际验证中体现出的优势包括:(1)验证模块化更明显;(2)验证代码的可拓展性更强.但本文的验证有一定的局限性,包括代码的精化关系未进行正式形式化验证;接口本身的可靠性未正式验证.这些局限性也是后续工作的一部分.

References:

[1] Wang J, Zhan NJ, Feng XY, Liu ZM. Overview of formal methods. Ruan Jian Xue Bao/Journal of Software, 2019,30(1): 33−61 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5652.htm [doi: 10.13328/j.cnki.jos.005652].

[2] Liang H, Feng X, Fu M. Rely-guarantee-based simulation for compositional verification of concurrent program transformations[J]. ACM Transactions on Programming Languages and Systems (TOPLAS),2014,36(1):1-55.[DOI:http://dx.doi.org/10.1145/2576235].

[3] Xu, FW, Fu, M, Feng XY., Zhang XR, Zhang H, Li, ZH. A practical verification framework for preemptive OS kernels [C]//International Conference on Computer Aided Verification. Cham: Springer International Publishing,2016:59-79.[doi:10.1007/978-3-319-41540-6_4].

[4] D. Ghosh, L. Tomazeli, F. Jin, M. Maheswaran. SpaceOS: Operating system services for smart computing environments, Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014. IEEE,2014: 1-6. [doi: 10.1109/WoWMoM.2014.6918932].

[5] Qiao L, Yang MF, Tan YL, Pu GG, Yang H. Formal verification of memory management system in spacecraft using Event-B. Ruan JianXueBao/JournalofSoftware,2017,28(5):1204−1220 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5218.htm [doi: 10.13328/j.cnki.jos.005218].

[6] Klein, G., Huuck, R. & Schlich, B. Operating System Verification. J Autom Reasoning. 2009,42(1):123–124. [doi:10.1007/s10817-009-9126-9].

[7] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, Simon Winwood. seL4: Formal verification of an OS kernel, Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. 2009:207-220.[doi:10.1145/1743546.1743574].

[8] Andronick, June & Lewis, Corey & Morgan, Carroll. Controlled Owicki-Gries Concurrency: Reasoning about the Preemptible eChronos Embedded Operating System. Electronic Proceedings in Theoretical Computer Science. 2015,196:10-24.[doi:10.4204/EPTCS.196.2].

[9] Galen C. Hunt and James R. Larus. 2007. Singularity: rethinking the software stack. SIGOPS Oper. Syst. Rev. 2007,41(2): 37–49.[https://doi.org/10.1145/1243418.1243424].

[10] Mohan Cui, Chengjun Chen, Hui Xu, and Yangfan Zhou. SafeDrop: Detecting Memory Deallocation Bugs of Rust Programs via Static Data-flow Analysis. ACM Trans. Softw. Eng. Methodol. 2023,32(4-82):1-21.[doi:https://doi.org/10.1145/3542948].

[11] Zhenyang Dai, Shuang Liu, Vilhelm Sjoberg, Xupeng Li, Yu Chen, Wenhao Wang, Yuekai Jia, Sean Noble Anderson, Laila Elbeheiry, Shubham Sondhi, Yu Zhang, Zhaozhong Ni, Shoumeng Yan, Ronghui Gu, and Zhengyu He. Verifying Rust Implementation of Page Tables in a Software Enclave Hypervisor. In Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2024,2(2):1218–1232.[doi:10.1145/3620665.3640398].

[12] J. Toman, S. Pernsteiner and E. Torlak, Crust: A Bounded Verifier for Rust (N), 30th IEEE/ACM International Conference on Automated Software Engineering (ASE),2015.3-9.[doi: 10.1109/ASE.2015.77].

[13] Kroening, Daniel, Michael Tautschnig. CBMC -C Bounded Model Checker(Competition Contribution).International Conference on Tools and Algorithms for Construction and Analysis of Systems. 2014.1-3.

[14] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. Leveraging rust types for modular specification and verification. Proc. ACM Program. Lang. 2019,3(147):1-30.[doi:10.1145/3360573].

[15] Johannes Schilling. Specifying and Verifying Sequences and Array Algorithms in a Rust Verifier[Matser. Thesis]. Switzerland-Zurich: Eidgenössische Technische Hochschule Zürich, 2021.

[16] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language[J]. Proceedings of the ACM on Programming Languages, 2017,2(POPL):1-34..[doi:10.1145/3158154].

[17] Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. GhostCell: separating permissions from data in Rust[J]. Proceedings of the ACM on Programming Languages, 2021,5(ICFP):1-30.[doi:10.1145/3473597].

[18] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code. In Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2022). 2022. 841–856. [doi:10.1145/3519939.3523704].

[19] Paulin-Mohring, C. Introduction to the Coq Proof-Assistant for Practical Software Verification. In: Meyer, B., Nordio, M. (eds) Tools for Practical Software Verification. LASER 2011. Lecture Notes in Computer Science. 2012,7682.[doi:10.1007/978-3-642-35746-6_3].

[20] Xavier Leroy. Formal verification of a realistic compiler. Commun. ACM. 2009,52(7):107–115. [doi:10.1145/1538788.1538814].

[21] Gonthier Georges. Formal Proof—The Four- Color Theorem. Semantic scholar. 2008. 1-12.

[22] David Delahaye, Micaela Mayero. Diophantus' 20th Problem and Fermat's Last Theorem for n=4: Formalization of Fermat's Proofs in the Coq Proof Assistant. 2005.hal-00009425.

[23] Jung R, Krebbers R, Jourdan J-H, Bizjak A, Birkedal L, Dreyer D. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. Journal of Functional Programming. 2018;28:e20.[doi:10.1017/S0956796818000151].

[24] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. In Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP). Association for Computing Machinery, 2016,256–269.[https://doi.org/10.1145/2951913.295194].

[25] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning[J]. ACM SIGPLAN Notices,2015,50(1):637-650.[doi:10.1145/2775051.2676980].

附中文参考文献:

[1] 王戟, 詹乃军, 冯新宇, 刘志明. 形式化方法概貌. 软件学报, 2019, 30(1): 33－61. http://www.jos.org.cn/1000-9825/5652.htm [doi:10.13328/j.cnki.jos.005652]

[5] 乔磊,杨孟飞,谭彦亮,蒲戈光,杨桦.基于 Event-B 的航天器内存管理系统形式化验证.软件学报,2017,28(5):1204−1220.http:// www.jos.org.cn/1000-9825/5218.htm [doi: 10.13328/j.cnki.jos.005218]